LA-MPI: The Design and Implementation of a Network-Fault-Tolerant MPI for Terascale Clusters.*

Robbie T. Aulwes, David J. Daniel, Nehal N. Desai,
Richard L. Graham, L. Dean Risinger and Mitchel W. Sukalski

Los Alamos National Laboratory
Advanced Computing Laboratory
MS-B287, P. O. Box 1663
Los Alamos NM 87545, USA

Categories and Subject Descriptors

B.8 [**Hardware**]: Performance and Reliability; C.2 [**Computer Systems Organization**]: Computer-Communication Networks; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Measurement Performance Reliability

Keywords

Message passing, fault tolerance, MPI

ABSTRACT

In this paper we discuss unique architectural elements of the Los Alamos Message Passing Interface (LA-MPI). LA-MPI is a high-performance, network fault-tolerant, thread-safe MPI library designed for terascale clusters that are inherently unreliable due to their sheer number of system components and inherent trade-offs between cost and performance. We examine in detail the design concepts used to implement LA-MPI. These include reliability features, such as application-level checksumming, message retransmission, and automatic message re-routing. Other key performance enhancing features, such as concurrent message routing over multiple, diverse network adapters and protocols, and communication-specific optimizations (e.g., shared memory) are examined.

*Email: `lampi-support@lanl.gov`. Los Alamos report LA-UR-03-0939. Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. Project support was provided through ASCI/PSE and the Los Alamos Computer Science Institute.

1. INTRODUCTION

One consequence of the rise of cluster and grid computing is the growing concern with fault tolerance of processors, communication networks, and system infrastructure. This is because the manufacturing tolerances to which such systems conform may be inadequate to guarantee error-free execution [13] of an application, given the length of a typical application run and the very large number of individual systems that are aggregated into a cluster. For example, a network device may have an error rate which is perfectly acceptable for a desktop system, but not in a cluster of thousands of nodes, which must run error free for many hours or even days to complete a scientific calculation.

Similarly to Bosilca [4], we divide messaging fault tolerance approaches into three levels depending on where in the software stack they are implemented. These three levels are: the upper level (e.g., application level), the mid level (e.g., transport level), and lower level approaches (e.g., data link level). Any software implementation may address fault tolerance at one or more of these levels. The levels at which a fault-tolerant mechanism operates implicitly makes assumptions about the type and probability of failures (or faults).

There have been a number of research efforts attempting to incorporate network and process fault tolerance into message passing systems. One of the first efforts to incorporate fault tolerance into MPI was CoCheck tuMPI [17] from Technischen University Munich, which addresses fault tolerance at an upper level. CoCheck used the Condor [8] library to checkpoint and then if necessary restart and rollback the MPI job. This system's main drawback was the need to checkpoint the entire application, which could be prohibitively expensive in terms of time and scalability for large applications (like those that would run on a terascale cluster). Another effort, Starfish MPI [3], is similar in operation to CoCheck, and also operates at an upper level. However, Starfish uses its own systems to checkpoint jobs, and does not rely on a flush message protocol to handle communications. Starfish uses "atomic" group communications protocols based on the Ensemble system [6]. A third upper level approach is the FT-MPI [5] effort from the University of Tennessee-Knoxville. FT-MPI handles fault tolerance at the MPI communicator level, and lets the application developer decide what course of action they wish to take. The

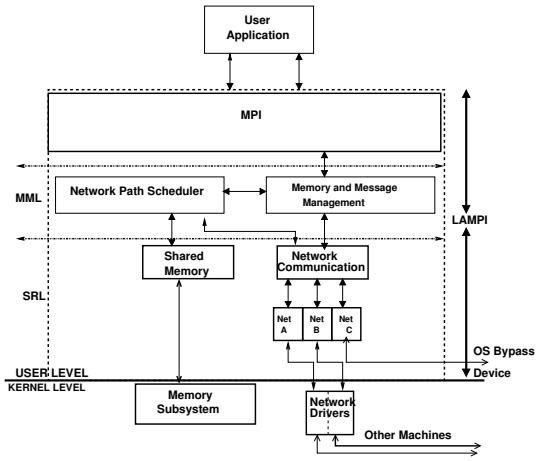

Figure 1: LA-MPI Architecture.

application may decide to shrink, rebuild or abort the communicator depending on the type of fault.

LA-MPI is an implementation of the Message Passing Interface (MPI) [11], the *de facto* standard interprocess communication API for scientific applications, in which we attempt to address fault tolerance at all of these levels. Our current efforts are directed at the lower and mid levels. It implements version 1.2 of the standard, and is integrated with ROMIO [18] for MPI-IO version 2 support. LA-MPI (a) reliably delivers messages in the presence of I/O bus, network card and wire-transmission errors; (b) survives network card and path failures (when the operating system survives) and guarantees delivery of in-flight messages after such a failure; (c) supports the concurrent use of multiple types of network interface; and (d) implements message striping across multiple heterogeneous network interfaces, and striping of message fragments across multiple homogeneous network interfaces. In the next section we motivate the architecture of the LA-MPI framework that is described in the rest of the paper. LA-MPI gives the MPI application developers a guarantee of end-to-end network-fault-tolerance and provides an excellent substrate for reliable applications.

This paper gives a detailed description of the architecture of LA-MPI, focusing on its fault-tolerant features.

2. ARCHITECTURE

Figure 1 shows the basic architectural elements of the LA-MPI library. LA-MPI provides MPI version 1.2 and some version 2 services by layering MPI routines on a basic set of User Level Messaging (ULM) interface primitives. ULM was designed to be a high-performance, fault-tolerant, reliable messaging subsystem capable of supporting any number of MPI implementations, and of being extended to support different messaging models (e.g., put/get one-sided communications, etc.).

ULM itself consists of two layers: the Memory and Message Layer (MML) and the Send and Receive Layer (SRL). The MML provides message management services including message routing (i.e., network path selection), message tag matching, buffer allocation (for uniform and non-uniform memory access machines (NUMA)), message retransmission, and message status tracking. The SRL provides message transmission and reception over shared memory and different network adapters. Each network type, or path, manages its own resources, and implements its own flow control and resource exhaustion schemes. All elements of ULM are designed to be non-blocking and thread-safe in operation.

Besides the basic architectural elements of the library, LA-MPI also implements a run-time system consisting of an executable, mpirun, and the MPI library for process spawning, standard I/O handling, job control, and network topology wire-up. To start a MPI job, LA-MPI spawns the user executable as a single process on each machine, specified either explicitly or through a resource reservation system such as Platform's LSF [15]. This single process upon calling *MPI_Init()* participates in local network resource discovery, which is shared globally as needed. After this first phase of network wire-up, this process forks itself to create all of the desired MPI processes. As a performance enhancement, forking is handled by default as a tree of processes. A second post-fork phase of network wire-up is then initiated and performed for those paths that require unique information from each MPI process (e.g., unique UDP ports for each MPI process). At the end of this second phase, all processes synchronize through a global barrier. On architectures in which LA-MPI handles standard I/O redirection and/or job control itself, the original process never exits *MPI_Init()*, and instead daemonizes itself to handle I/O redirection, signal handling, and job termination.

2.1 MPI

By layering MPI on ULM, most MPI routines can be implemented in three phases: argument checking, dispatch to the appropriate ULM routines, and error code and status translation, if necessary. A function dispatch table provides a means of selecting different implementations of collective operations at run-time, to provide optimal performance in different environments. This basic layering approach allows modularity, thereby facilitating code maintenance, while minimizing the impact upon performance-critical services. A compile-time flag controls whether MPI parameters are validated before use, and the extra performance gained can be balanced against the users' confidence in the correctness of their MPI-dependent application code.

MPI communicators and groups are shared between the MPI and ULM layers. They have been augmented to provide functionality not strictly required by MPI semantics, but needed for internal operation and performance optimizations (e.g., precomputed and cached information to reduce the execution time of repeated code). For example, the group object has extra maps, such as a mapping of process rank to local process rank (a unique rank between 0 and n-1 for n local processes on a given machine) for faster access into local arrays of lists. Also, precomputed trees are associated with each group to enhance the performance of certain collectives (i.e. gather/scatter operations between multiple machines). *ulm_get_info()* and a number of other ancillary functions are used by the MPI layer to access information in these ULM objects.

MPI data types are stored as a "flattened" array of (offset, size, sequential_offset) structures. Contiguous offsets are merged into a single entry, and a data type with contiguous, in-order offsets will be represented by a single structure. This flattened representation is highly efficient for data packing and unpacking. It is also relatively memory-efficient for many typical data types, but it does not support a single MPI job over machines with different byte ordering (i.e., a mixture of big-endian and little-endian machines), which is rare in most cluster environments. Constructor information from MPI type calls, such as *MPI_Type_Struct()*, is saved for *MPI_Type_get_envelope()* and *MPI_Type_get_contents()* (MPI version 2) functionality. This information is traversed recursively as needed for *MPI_Get_elements()*. Reference counting is implemented for all data types to avoid the unnecessary overhead of creating duplicates of existing data types for *MPI_Type_get_contents()*.

2.2 Memory and Message Layer

The Memory and Message Layer (MML) is a set of common abstractions to ensure reliable message delivery and matching while minimizing the overhead of memory management of LA-MPI data structures and buffers. In short, it is the common code between MPI functionality and network transport-specific code (i.e., the SRL).

User request objects, message send descriptors, and message receive descriptors are managed by the MML (figure 2). User request objects track MPI send and receive parameters, such as the message tag, source or destination process, communicator, etc. The user request object also contains completion status information and a flag that indicates whether the operation has completed. The message send descriptor contains all of the necessary information to send a message reliably, including lists of message fragment descriptors. The message receive descriptor is used for matching message fragments with a particular message; the receive descriptor also keeps track of the number of bytes received and discarded due to a too small receive buffer.

Some of these objects, as well as other SRL objects, are stored in arrays of free lists. Each free list may be constructed with memory policies to allocate machine memory close to a given process rank's processor and its local memory. This architectural consideration is extremely important for good performance on large NUMA machines, such as SGI's Origin 2000 and 3000 machines. Memory allocation for long-lived objects is managed with sub-allocators that subdivide large chunks of memory, allocated via *malloc()* or *mmap()*, into fixed elements whose addresses are quickly allocated and released by the use of a simple stack [9]. If the memory is private to a process, then new elements can be created by allocating another chunk of memory. If the memory is anonymous shared memory (allocated from a common ancestor and hence growable only with local process synchronization), then resource exhaustion is handled with simple retry logic until a maximum number of retries has been reached. Experience on a number of platforms has shown that the sub-allocator approach is significantly faster than a general-purpose allocator, such as *malloc()*.

User request objects are separate from send and receive descriptors to support persistent requests efficiently without

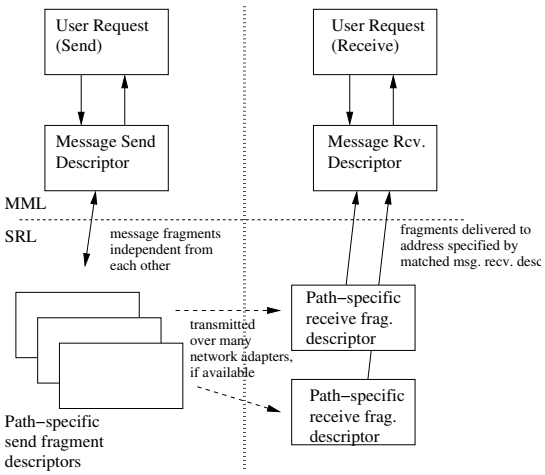

Figure 2: Descriptor Relationships.

replicating the request object. Persistent send requests can generate multiple message send descriptors that must be managed concurrently by the MML, because of MPI's local send completion semantics. These semantics are extremely important for low-latency performance in most typical MPI applications, because the true overhead of a blocking send as a result of waiting for fragment acknowledgments can be hidden for those messages whose contents are buffered by LA-MPI.

All messages are transmitted in fragments whose size is determined by the underlying SRL network transport (or path, as described later in section 2.2.2). Each network transport handles its own flow control to prevent resource exhaustion, and minimize the memory footprint of the library as a whole. Message fragmentation also allows LA-MPI to transmit a message simultaneously over different physical interfaces for a given network transport. For example, on LANL machines made up of multiple SGI Origin 2000 SMPs interconnected by a switched network with 2 to 12 HIPPI-800 adapters each, LA-MPI can effectively transfer an MPI message over all 2 to 12 adapters simultaneously for greater bandwidth.

Each message send descriptor manages two lists of these fragment descriptors: *FragsToSend* for fragments whose resources still need further allocation, and *FragsToAck* for sent fragments that are awaiting acknowledgment of successful delivery from a peer process. Each message send descriptor itself is stored on one of two similarly paired lists: an incomplete list for send descriptors that still need further processing for all fragments to be sent, and an unacknowledged list for send descriptors awaiting peer acknowledgment.

Message progress from incomplete to unacknowledged for fragments and all message descriptors is made via calls to *ulm_make_progress()*. These calls are embedded in MPI operations such as wait and test, and other potentially blocking calls such as *MPI_Barrier()*. Currently, LA-MPI does not utilize a separate thread for making progress, but uses polling in various library entry points. The indiscriminate use of threads can lead to an oversubscription of proces-

sors, which creates resource contention. Polling prevents this cache and context switching contention with the application's main code, but does require special design considerations to handle the uncertainties of MPI scheduling. In particular, the retransmission scheme must be designed not to overrun receiver resources simply because the remote process is doing non-MPI processing at the moment.

2.2.1 Fragment Retransmission and Checksumming

Unlike many MPI libraries that consider all underlying communication perfectly reliable, LA-MPI optionally supports sender-side retransmission of messages by checking the unacknowledged list every 5 seconds (adjustable at compile time) for message send descriptors that have exceeded their timeout periods. This retransmission scheme is appropriate for low error rate environments, typical of most clusters. Each network transport is responsible for arranging to retransmit the necessary fragments. Each fragment's retransmission time is calculated using a truncated exponential back-off scheme; this avoids resource exhaustion at a receiving process that is busy doing non-MPI computation. Fragments that must be retransmitted are moved from the *FragsToAck* list to the *FragsToSend* list, and the associated message send descriptor is placed on the incomplete list.

Each network transport is also responsible for providing a main memory-to-main memory 32-bit additive checksum or 32-bit cyclic redundancy code (CRC), if it is needed. This checksum/CRC protects against network and I/O bus corruption, and is generated at the same time data is copied, if at all possible. By delaying checksumming to avoid wasting memory bandwidth, a received fragment is not necessarily a deliverable, or uncorrupted, fragment.

Several MML generic features aid in the implementation of this retransmission and checksumming scheme. Every fragment is assigned a monotonically increasing 64-bit sequence number between a given (sender, receiver) pair of processes. These sequence numbers are recorded by the receiving process in a special object, *SeqTrackingList*, as an ordered set of non-contiguous ranges of sequence numbers; these lists use internal hint pointers to exploit any temporal locality in accessing these lists to minimize access overhead. The receiver maintains two *SeqTrackingList* lists for each peer with which it communicates to distinguish between fragments that have been received, and those that have been received and delivered successfully (i.e., no data corruption). Duplicate fragments are easily detected by checking the received fragment's sequence number against the received *SeqTrackingList*.

Upon processing fragment acknowledgments from a receiver, a sender will store two special values that are carried in every acknowledgment: the largest in-order peer received sequence number (LIRS), and the largest in-order peer delivered sequence number (LIDS). The LIRS is used to prevent the retransmission of fragments that have been received, but whose data integrity has not been checked yet; it may increase or decrease over time, depending upon transmission and I/O bus errors. The LIDS is used to free any fragments whose acknowledgment was lost. The LIDS is always less than or equal to the LIRS. Figure 3 shows the interaction of these sequence numbers, the retransmission scheme, and

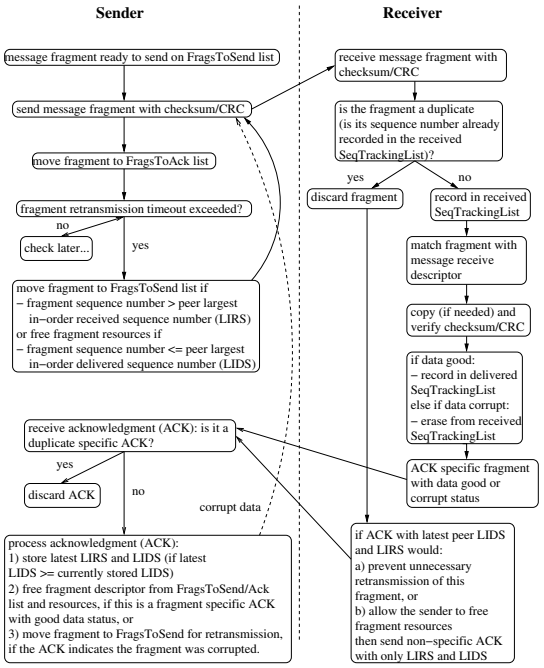

Figure 3: Retransmission and Checksumming.

checksumming.

2.2.2 Network Path Interactions

The network path object is an abstraction of lower-level network transports and devices that are available to LA-MPI. Each path can represent a single network adapter, or a set of common adapters, or even a common protocol over many different network adapters. Currently, paths are implemented for the user datagram protocol [16] (UDP over all IP-enabled devices), Quadrics Elan3 [14] remote direct memory access (RDMA), and HIPPI-800 operating system bypass (SGI IRIX only). Paths are currently being developed for Myrinet GM [12], and Infiniband [7] (Mellanox HCA Verbs [10]). In all of our current paths except UDP/IP, which treats multiple network adapters as a single Internet Protocol "device", multiple network adapters are used by a single path instantiation, if they exist on the machine. By fragmenting messages and sending different fragments over different network adapters, LA-MPI takes advantage of all of the available bandwidth.

Each path provides a common set of services to the library (via virtual methods):

- to store/retrieve a handle to a path: *bindToContainer()*, *getHandle()*, and *getContainer()*;
- to control a path's status and query remote process reachability: *isActive()*, *activate()*, *deactivate()*, and *canReach()*;
- to store/retrieve information about a path: *getInfo()*, and *setInfo()*;
- to control the ownership of a MPI message: *bind()*, and *unbind()*;

- to initialize and send/resend a bound message: *init()*, *send()*, *retransmitP()*, and *resend()*;
- to query a message's send status: *sendDone()*;
- to receive message fragments from a path: *receive()*; and
- to check whether control messages (non-MPI message data) need sending and to send those control messages: *needsPush()*, and *push()*.

Two functions stored in each MPI communicator provide network path selection on a per-MPI message basis. One function controls the binding of point-to-point messages, and the other of multicast messages. These functions can be manipulated on a per-communicator basis via a get/set ULM interface. The default functions implement a static hierarchy of network paths based on the nominal bandwidth; if there is only one path available, the function returns immediately with that one.

At startup, each path does basic initialization, and registers itself with a global *pathContainer*. During registration the pathContainer queries the path to determine which processes it can reach via the path. One method, *paths()*, is used to find all of the currently active paths to a given process, and another, *allPaths()*, is used by *ulm_make_progress()* to poll all active paths for received data.

Figure 4 illustrates the interactions between send messages and paths. Using these abstractions, LA-MPI supports automatic network fail-over in the face of general and specific network failures, including network adapter, switch, and link failures. The normal cycle of path selection, path *bind()*, *init()*, and *send()* is interrupted by a network failure. Many network transports do not provide explicit failure notification, so timeouts are used in these cases. Timeouts are either relatively small for paths that signal data link layer completion, such as Quadrics Elan3 RDMAs, or, in the face of no information, much larger to account for maximum acknowledgment periods.

Each path sets its own policy for determining when a path failure has occurred, and signals that a message needs to be sent over another path (i.e., rebound to another path) by returning a path failure error code. If a path manages multiple network adapters, and another managed adapter can reach the receiving process, then the path is responsible for silently retransmitting the affected traffic over that adapter without signaling failure.

If path failure is signaled and the message rebinds successfully, the new path uses a special synchronization protocol to determine which of the previous path's fragments were actually received using the fragments' unique sequence numbers (discussed in section 2.2.1). The receiving process notes which of the sending process' outstanding fragments have been received and/or delivered. It purges those fragments that have been received but not delivered, and responds to the sending process with a list of the fragment numbers that have not been delivered. Finally, it marks all of the outstanding "old" sequence numbers as having been received and delivered (to prevent "old" fragments that might still be received from being processed). The sending process then transmits the missing fragment data over the new path using new fragments with new unique sequence numbers (since there is no guaranteed correspondence between the two paths' fragment sizes).

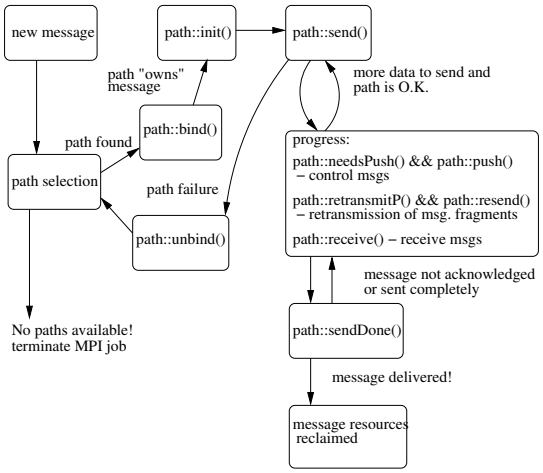

Figure 4: Message-Path Interactions.

2.2.3 Message Tag Matching

Because LA-MPI supports the concurrent use of multiple networks, message tag matching is implemented independently of the underlying networks. In order to preserve MPI semantics, messages between any two processes must be matched by the receiver in the order that they are posted by the sender. LA-MPI accomplishes this with a message sequence number that is sent with all fragments of a message. Since fragments can be received out-of-order over different network adapters, tag matching can be accomplished with any of a message's fragments. This sequence number increases monotonically, and is unique between a given sending and receiving process. Each receiving process keeps track of the next expected sequence number for a given sending process. While this sequence number prevents matching messages out-of-order from the sending process' perspective, message receive descriptors are kept in the order they are posted through the use of an ordered list to which a descriptor is always appended.

When a message fragment is received, its message sequence number is either greater than, equal to, or less than the next expected sequence number. These conditions correspond respectively to receiving a fragment ahead of sequence, receiving the first fragment of a message that should be matched immediately, and receiving a fragment for a message that has already been matched if at all possible. If a fragment's message sequence number is greater than the expected sequence number, then the fragment is stored on a special array of lists for ahead-of-sequence fragments that are indexed by the sending process' rank.

If a fragment's message sequence number is equal to the expected sequence number, the fragment is matched against the next appropriate message receive descriptor. If a match

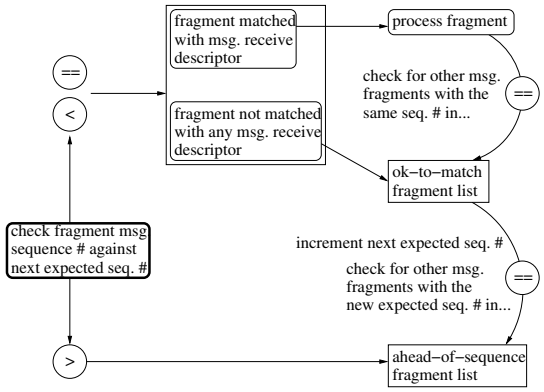

Figure 5: Message Tag Matching.

is not made, then the fragment is stored on a special ok-to-match array of lists that is indexed by the sending process' rank. If a match is made, then the fragment is processed (copied, checksum/CRC verified, and an ACK generated, as needed), and any fragments of this same message on the ok-to-match list are also processed. The matched message receive descriptor is moved to a special list for matched descriptors. In either case, the receiver then increments the next expected sequence number, and checks the ahead-of-sequence fragment list for a possible match to the new expected sequence number.

Finally, if a fragment's message sequence number is less than the next expected sequence number, then the fragment must belong to a message whose first fragment has already arrived. This fragment is processed in almost the same way as the sequence equality case. If a match was already made by checking the special matched descriptor list, then the fragment is processed. If a match has not been made, then the fragment is appended to the ok-to-match list, along with the first fragment and any other fragments that have previously arrived. Figure 5 illustrates the entire matching process.

In multi-threaded operation LA-MPI uses an array of receive locks, indexed by the sending process rank, to prevent race conditions from the simultaneous posting of message receive descriptors (e.g., *MPI_Irecv()*), and processing of received fragments. In processing non-wild-card receive requests, only threads trying to receive messages from the same sending process must contend for the same locks. Wild-card requests, however, require the acquisition of all receive locks; the resultant performance degradation is a design trade-off against the performance gained for non-wild-card requests.

2.3 Send and Receive Layer

The Send and Receive Layer (SRL) consists of multiple network path implementations and a highly optimized shared memory communication implementation. Each implementation is independent of the others, and optional run-time controls can specify which paths should be used, even if others are available.

2.3.1 Shared Memory

LA-MPI's shared memory communication uses shared memory for the message send descriptor, the message fragment descriptors, and data. This means that the receiving process receives 32- or 64-bit pointers to these control structures, and does not incur the overhead of allocating and describing receive fragments. The only cost of "data transmission" is remote access to another processor's local memory, which can be relatively considerable on large NUMA machines, especially with cache coherency. LA-MPI's memory locality support is used, however, on NUMA machines with the appropriate support to make sure the control structures and data are allocated close to the sending process' processor. This ensures that at least one of the processes has minimum latency to the shared memory.

One of the key shared memory communication structures is a two dimensional array of first-in-first-out (FIFO) lists indexed by the sending and receiving process' ranks. This array of FIFOs is constructed to use memory near the sending process, and is used to transmit the 32- or 64-bit addresses of fragment descriptors to a receiving process. Since each FIFO has only one writer, namely the sending process, cache invalidation thrashing by multiple processors writing to the same memory is avoided.

The first fragment descriptor of a message (*SMPFragDesc_t*) is different from all other fragment descriptors (*SMPSecondFragDesc_t*); the first fragment descriptor contains all of the information needed to match the message to a posted receive descriptor at the receiving process. Once the first fragment is matched, no further matching is required for that message. The receiving process puts a pointer to its message receive descriptor in the first fragment descriptor and the message send descriptor. If a match has already been made, then the following fragments of a multi-fragment message are placed on the receiver's *SMPMatchedFrags* list, with a pointer to the message receive descriptor stored in the fragment descriptor. Otherwise, they are placed on a *fragsReadyToSend* list in the message send descriptor so that they can be properly processed by the receiver upon making a match.

Since all messages need at least one fragment descriptor, allocation costs are minimized by creating message send descriptors and first fragment descriptors as adjacent, paired objects. They are then allocated as a single object. As a further optimization, the first fragment descriptor can be created, and its address transmitted via the fixed FIFO (and an overflow list) to a receiver before the sender actually copies message data into shared memory. The receiver's cost of matching the first fragment to a message receive descriptor can be hidden by the time required for the sender to copy data into shared memory. This is a performance gain for all messages greater than zero bytes in length.

2.3.2 Network Communication

Each network path has its own design considerations in order to balance performance, scalability, and memory usage. In this section we discuss interesting design points for different network transports and devices without covering them in exhaustive detail, for brevity's sake.

LA-MPI's user datagram protocol (UDP/IP) path implementation uses two UDP sockets per MPI process. To sup-

port full mesh connectivity between N processes, LA-MPI needs 2N UDP connection-less sockets, as opposed to N(N-1)/2 connection-oriented sockets. This scalability becomes very important in terascale distributed computing environments with thousands of MPI processes, and was a major factor in choosing UDP/IP over connection-oriented TCP/IP.

The UDP/IP implementation's two sockets are used for two different types of traffic. The short message socket supports eager sending of small messages, the first fragment of a multi-fragment message, and control messages such as fragment acknowledgments. The long message socket supports fragments of multi-fragment messages that have already been matched. A multi-fragment message's first fragment is sent to the receiving process' short message socket. When it is matched and processed, an acknowledgment is generated and sent back to the sending process. Upon receiving the acknowledgment, the sender then sends the remaining fragments to the receiver's long message socket. This protocol ensures that any traffic on the long message socket can be processed immediately without additional user-level buffering that must be maintained for an indefinite period of time.

All UDP/IP message fragments have a fixed MPI header that is peeked at first (i.e., copied but not deleted from kernel buffers), so that if the fragment's contents represent contiguous MPI data types, then the data can be placed from kernel memory to the proper user address via one *recvmsg()* call.

LA-MPI's Quadrics Elan3 RDMA implementation uses a radically different approach from the UDP/IP implementation. The Elan3 library provides facilities to create sequential chains of actions using events, and support for two types of RDMA writes into a remote process' 32-bit address space: a queue DMA that delivers a payload up to 320 bytes to a remote queue whose access is coordinated between the network adapter and the main processors; and a normal DMA that delivers an arbitrary payload to a particular virtual memory address of a remote process. A block copy event copies the contents of an Elan adapter memory region to another Elan-addressable region, usually in main memory; by chaining a block copy event at the end of a set of RDMA writes, a local process can be notified that all of the previous RDMA operations have completed. This is extremely useful, if the ACK protocol has been disabled by run-time settings, or if the process is trying to detect path failure.

LA-MPI utilizes both of these types of DMAs by sending fragments' header information (with 52 bytes of immediate data payload) and all control messages via queue DMAs with a queue element size of 128 bytes, and by sending larger fragments via normal DMAs to Elan3 addressable memory that has been allocated using a memory request/response protocol. Fragments with 53 bytes of data or more are sent using chained DMAs, with the data sent first using a normal DMA and the header information sent next using a queue DMA. A special copy event that copies a known value into an event block in main memory terminates the chain, and is used to signal the completion of the chained DMA locally.

Each process allocates a single queue per network adapter,

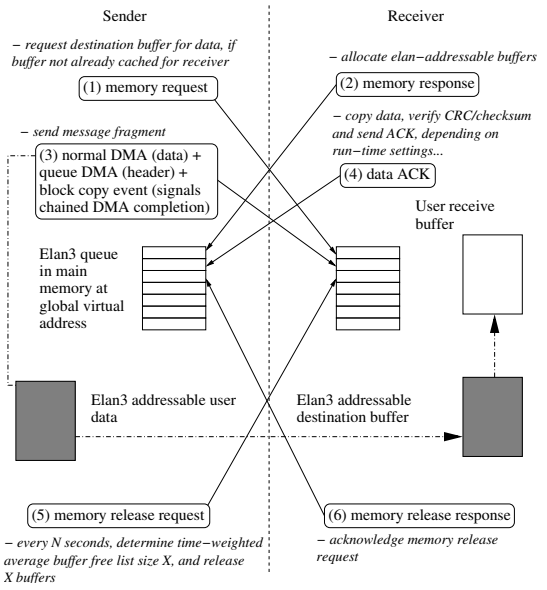

Figure 6: Quadrics Elan3 Send and Receive Implementation.

or rail in Quadrics parlance, at the same virtual memory address. By sending all control information to one of these queues, LA-MPI minimizes polling overhead. The Elan3 adapter only supports access into a 32-bit virtual memory address space; on 64-bit architectures the Elan3 adapter can only access a static portion of the entire process address space. Since not all memory is guaranteed to be addressable, a special set of control messages is used between a (sender, receiver) pair of processes to request elan-addressable memory buffers. The sender caches these addresses for later use, and uses them in a last-in-first-out (LIFO) fashion to reduce the chances of a page fault. The sender maintains a time-weighted average free list size of these cached addresses, and lazily releases the average number of buffers on the free list to the receiver process every couple of minutes. Figure 6 illustrates the basic send and receive mechanisms.

3. SUMMARY AND FUTURE WORK

With the rise of terascale distributed computing environments consisting of thousands of processors and network adapters, the need for fault tolerant software has become critical to their successful use. Negligible component error and failure rates in small to medium size clusters are no longer negligible in these large clusters, due to their complexity, sheer number of components, and amount of data transferred. LA-MPI addresses the network-related challenges of this environment by providing a production-quality, reliable, high-performance Message Passing Interface (MPI) library for applications capable of (a) surviving network and I/O bus data corruption and loss, and (b) surviving network hardware and software failure if other connectivity is available. In this paper, we have presented an overview of LA-MPI's design and implementation.

LA-MPI is currently available as open source software under an LGPL license. It currently runs on Linux (i686 and Alpha processors), HP's Tru64 (Alpha only), SGI's IRIX

6.5 (MIPS), and Apple's Mac OS X (PowerPC). It supports shared memory, UDP/IP, Quadrics Elan3 RDMA, HIPPI-800 OS bypass (IRIX only), and current work is progressing on Myrinet GM and Infiniband (Mellanox HCA Verbs) communications support. LA-MPI supports job spawning and control with Platform LSF, Quadrics RMS (Tru64 only), Bproc [1], and standard BSD rsh. Please send email to `lampi-support@lanl.gov` for more information [2]. All fault tolerance features described in this paper have been fully implemented, except for on-going work on automatic network fail-over support.

Future papers will present full performance studies of design trade-offs as part of our on-going optimization of the library. Future development efforts will address:

- the implementation of a fault-tolerant, scalable, administrative network for job control, standard I/O redirection, and MPI wire-up;
- the implementation of process fault-tolerance in the face of multiple process failures;
- the implementation of dynamic topology reconfiguration and addition of MPI processes to support dynamic process migration and MPI-2 dynamic processes.

4. REFERENCES

[1] Advanced Computing Laboratory, Los Alamos National Laboratory. *http://public.lanl.gov/cluster/index.html*.

[2] Advanced Computing Laboratory, Los Alamos National Laboratory. *http://www.acl.lanl.gov/la-mpi*.

[3] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[4] G. Bosilca, A. Bouteiller, F. Cappello, S. Djailali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes.

[5] G. Fagg and a Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *EuroPVM/ MPI User's Group Meeting 2000, Springer-Verlag, Berlin, Germany, 2000*, 2000.

[6] G. E. Fagg, K. Moore, and J. J. Dongarra. Scalable Networked Information Processing Environment (SNIPE). *Future Generation Computer Systems*, 15(5–6):595–605, 1999.

[7] Infiniband Trade Association. *http://www.infinibandta.org/home*.

[8] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *8th International Conference on Distributed Computing System*, pages 108–111. IEEE Computer Society Press, 1988.

[9] M. McKusick and M. Karels. Design of a general purpose memory allocator for the 4.3 BSD Unix kernel. In *Proceedings of the Summer 1988 USENIX Conference*, pages 295–303. USENIX Association, 1998.

[10] Mellanox Technologies, Inc. *http://www.mellanox.com/*.

[11] Message Passing Interface Forum. MPI-2.0: Extensions to the Message-Passing Interface. Technical report, 1997.

[12] Myricom, Inc. *http://www.myri.com/*.

[13] C. Partridge, J. Hughes, and J. Stone. Performance of checksums and CRCs over real data. *Computer Communication Review*, v. 25 n. 4:68–76, 1995.

[14] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, v. 22 n. 1:46–57, 2002.

[15] Platform Computing, Inc. *http://www.platform.com/*.

[16] J. Postel. RFC 768: User Datagram Protocol, Aug. 1980.

[17] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[18] R. Thakur, W. Gropp, and E. Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*. Mathematics and Computer Science Division, Argonne National Laboratory, Oct. 1997. ANL/MCS-TM-234.